CS 330 – Spring 2023 – Homework 02

Due: Wednesday 2/8 at 11:59 pm on Gradescope

Collaboration policy Collaboration on homework problems is permitted, you are allowed to discuss each problem with at most 3 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and pseudocode (*),
2. a proof of correctness,
3. an analysis of the asymptotic running time of *your*.

You may use anything we learned in class without further explanation. This includes using algorithms from class as subroutines, stating facts that we proved in class, e.g. correctness of subroutines, running time of subroutines and use the notation. Your description should be at the level that it is clear to a classmate who is taking the course with you.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

(*) It is fine if the English description concentrates on the high level ideas and doesn't include all the details. But the reader should not have to figure out your solution solely based on the pseudocode. You can also add comments to your pseudocode, in fact that is best practice.

- By: Gianna Sfrisi U77992006
- Collaborators: Albert Slepak U00956163

Problem 1. Let $G(V, E)$ be a connected, undirected graph and let $s \in V$ be a source node (you may assume that the graph is given in an adjacency list format). For any $v \in V$, we call path from s to v *shortish* if it is either a shortest path or it has one more edge than a shortest path.

(a) Consider the BFS algorithm. Suppose that u is the next node that the algorithm explores its neighbors. Prove that for any neighboring node $v \in G[u]$ such that $dist_s(v) \leq dist_s(u)$, we have

$$dist_s(v) = dist_s(u) - 1 \quad \text{or} \quad dist_s(v) = dist_s(u),$$

where $dist_s(a)$ denotes the distance between s and some node $a \in V$ (this is, the length of the shortest path between s and a).

Proof by Induction:
In this problem, u's neighbors are the next nodes the algorithm will explore. By this sentence, u is stated to already of been discovered by the algorithm. In BFS, in order for a node's neighbors to be explored that node must have already been discovered. Every undiscovered node in a tree is on layer (n+1) if the previous node was found on layer n. In this example, if u is discovered on layer n (as its neighbors are being explored next) by induction, the next layer would be (n+1). If n is found true, (n+1) should also be true. The only contradiction to this statement is if the neighboring node was previously found through a different path to the source node. In this case the node would be on layer n. The problem does not specify if u's neighbors have been discovered, however in both scenarios node v is either on the same layer or one layer lower then u which supports the statement above.

(b) Design an algorithm that returns all nodes $v \in V$ that are connected to s with at least two distinct shortish paths. Prove the correctness of your algorithm and analyze its run time. Your algorithm should run in time $\Theta(n+m)$ where $n = |V|$ and $m = |E|$.

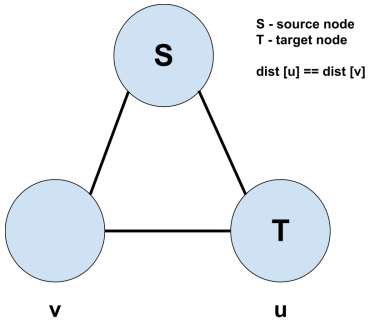
Algorithm:

```
/* G is hash table, the adjacency list of a graph */
/* s is a source vertex in G */
resultingNodes = {} /* empty hash table of nodes that the function will return that have a shortish
path */ parents = {}/* empty hash table, parents[v] = v's parent. */
dist = {}/* empty hash table, dist[v] = distance from s. */

Q = empty FIFO queue /* keep track of active nodes */
Q.enqueue(s), parents[s] = None, dist[s]=0 /* initialization */

while Q is not empty do
        u = Q.dequeue();
        for v in G[u] do
            /* explore neighbors of active node u */
                if v not in parents then


                    /* v was so far undiscovered */
                        parents[v] = u;


                    dist[v] = dist[u] + 1;
                        Q.enqueue(v);


                else
                    /* If there is an edge to another node on the the same layer, then there is another
shortish path to node u */
                    if dist[v] == dist[u] then
                            resultingNodes.add(u)
                    else if (v != parents[u]) then
                            if (dist[v] == dist[u] - 1) then
                                    resultingNodes.add(u)
                    else if (resultingNodes.contains(v)) then
                            resultingNodes.add(u)
return resultingNodes
```

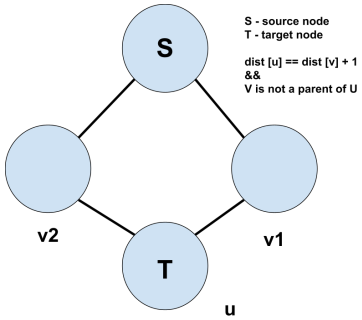
Run-time Analysis:

This algorithm runs in time $\Theta(n+m)$. The initialization in the first few lines of the code runs at constant time. The algorithm is a modified version of the BFS with additional if statements to account for the different cases. While Q is not empty the algorithm will add to the queue in the form of a while loop. That loop runs at time $\Theta(n)$ as it will loop through every node. Embedded in the while loop is a for loop that checks the different locations and relations to the parent node. Since it loops through every edge, the inner loop runs at time $\Theta(m)$. Therefore, combining the two loops into a singular algorithm the total run time is $\Theta(n+m)$.

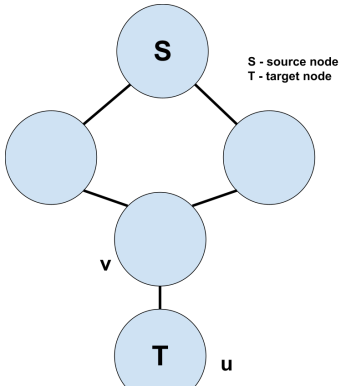
Proof of Accuracy:
The goal of this algorithm is to find the nodes that connect to the source node s with at least two distinct shortish paths. There are three different cases that must be accounted for when going about this problem.



(a) when dist[u] == dist[v]



(b) when dist[u] == dist[v] + 1 and v is not a parent of u



(c) When the other cases fail, but v is in the resulting list, so u should be as well

Figure 1: Illustration of Problem 1B.

As seen in Figure 1, the first case the algorithm accounts for is when there are two nodes connected by an edge on the same layer as each other (share a parent node). This case is accounted for in the first if statement and adds the nodes into the set of resulting Nodes. The second case represented in Figure 1 is when the distance between two nodes is one layer away. There are two different distinct paths to get from S to u: S $->$ v2 $->$ u and S $->$ v1 $->$ u. If neither of these cases apply, but v is still in the resulting list u should be included as well; this is shown in the third image of Figure 1. This algorithm is correct as it accounts for all of the possible cases for there to be two distinct paths between the source node s and any given node. It also has a run-time of $\Theta(n+m)$ following the guidelines of the original directions for the problem.

Problem 2. Let $G(V, E)$ be a directed graph and let $s \in V$ be a source node (you may assume that the graph is given in an adjacency list format). We say that a node is *reachable* from s if there is a directed path to it. (see illustration). Design an algorithm that finds and returns all nodes $v \in V$ such that there exists, in G, both a path from s to v **and** a path from v to s. The input to you algorithm is G and s and the output the set of nodes. Prove the correctness of your algorithm and analyze its run time. For full credit, your algorithm should run in time $O(n+m)$ where $n = |V|$ and $m = |E|$. For 8/10 pts, your algorithm can run in $O(n^2 + nm)$ time. *Note that for BFS it doesn't matter whether the input is the adjacency list of a directed or undirected graph.*

Algorithm:

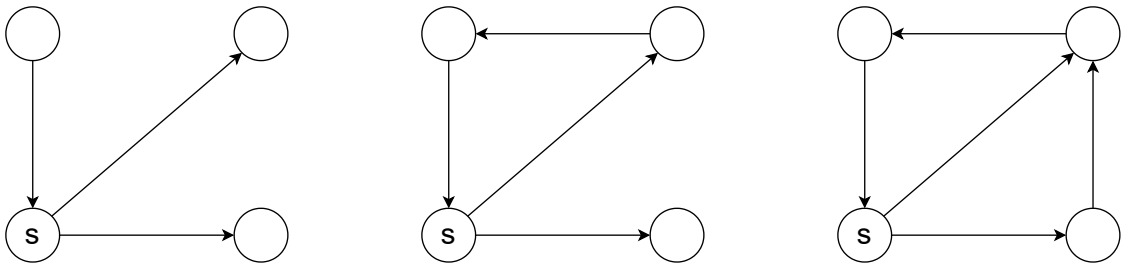
```
/* G is hash table, the adjacency list of a graph */
/* s is a source vertex in G */
resultingNodes = {} /* empty list of nodes that the function will return that have a shortish path */

/* Referencing the DFS algorithm from lecture. */
forwardParents, forwardDist = DFS(G,s);

/* Next we reverse the edges in the graph */
H = {} /* H is hash table, the reverse adjacency list of a graph */
for each vertex in G do
        for each neighbor in G[vertex] do
                H[neighbor].append(vertex)
/* Now we get reversed distances */
backwardsParents, backwardsDist = DFS(H, s);

for v in G do
        if (forwardDist.contains(v) and
backwardsDist.contains(v)) then
                resultingNodes.append(v);


return resultingNodes;
```

Run-time Analysis:

This algorithm has a run-time of $\Theta(n+m)$. The algorithm calls DFS twice, once on the original adjacency list and a second time on the reversed list. Each time DFS is called there is a run-time of $\Theta(n+m)$. Additionally, the nested for loop in the algorithm that reverses the original list runs at a time of $\Theta(n)$. The entire algorithm together would run at a speed of $\Theta(n+m) + \Theta(n+m) + \Theta(n)$ which would simplify to $\Theta(3n+2m)$. Therefore since the constants do not have a large impact on run-time, the overall time is $\Theta(n+m)$.

Proof of Accuracy:

The goal of this algorithm is to find all the nodes that have a path to and from the source node s. The first part of this problem is finding which nodes have a path from the source node. In order to find these nodes, the algorithm calls DFS on the original adjacency list, represented as variable G. The second part of the problem is finding if there is a path in the reverse direction, hence the path is bidirectional. The next part of the algorithm reverses the original adjacency list. In the algorithm, this is stored and represented by variable H. DFS is then called on the new adjacency list H which produces a new list of nodes connected to the source from the reverse direction. Lastly, the algorithm compares which nodes have a path in both directions and returns the set of nodes. The algorithm successfully finds which nodes have a path to and from the source node at a run time of $\Theta(n+m)$ which solves the problem.



(a) Some nodes are reachable in one direction.

(b) Some nodes are reachable in both directions.

(c) All nodes are reachable in both directions.

Figure 2: Illustration of Problem 2.